8. Hash Tables

- Motivation
- Hash Tables
- Integer Hashing
- String Hashing
- Storage and Retrieval
- Hashing Collisions
- Linear Probing
- Double Hashing
- Separate Chaining
- Hash Buckets
- Hashing Analysis
- Hashing Discussion

Motivation

- So far we have seen a variety of ADTs for storing and retrieving data.
- Goal of Hash Tables is to get faster access!

ADT	**Search Time**
array	O(N) – unsorted O(logN) – sorted
linked list	O(N) – sorted or unsorted
stack	O(1) – for top only
queue	O(1) – for head/tail only
binary tree	O(logN) – for any value
heap	O(logN) – for largest value

Hash Tables

- A hash table is a data structure invented for very fast data storage and retrieval.
- Goal is to look for data in only ONE step using the data itself to tell you where to look.
- We use a <u>hash function</u> to map data values into table positions.

  eg: hash("john") = 42 so we store "john" in position 42.

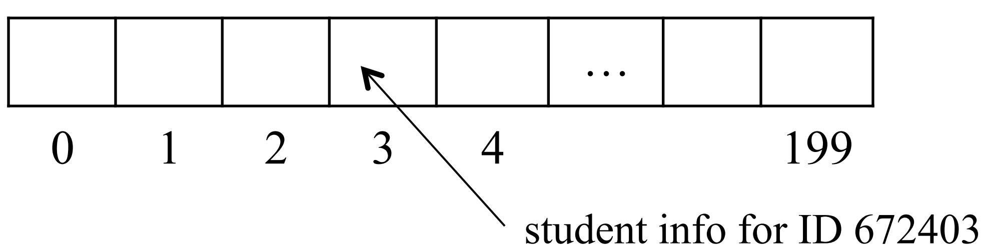
			…	john	…		
0	1	2		42			

- Hash functions are many to one, so we need an algorithm to resolve collisions (where 2 values map to the same table position).

Integer Hashing

- Assume that we want to store/retrieve 100 student records by their ID.
- We could allocate an array 1,000,000 long use ID as the index, but this would waste a lot of space.
- Instead, we can allocate an array 200 long and use ID % 200 as our hash function.

eg: hash(672403) = 3



- We need to make sure the hash table size is larger than number of records we intend to store.

String Hashing

- Goal is to spread index values uniformly around the hash table to reduce collisions.
- There are many ways to do this with a string.
  - add the ASCII codes for all characters in the string.
  - convert string into a number base 256.
  - select k characters from string and multiply ASCII codes by user defined position weights.

    eg. index = (str[2]*17 + str[3]*21 + str[6]*33) % size

j	o	h	n		g	a	u	c	h
0	1	2	3	4	5	6	7	8	9

Storage and Retrieval

- To store a record in hash table:
  - calculate index.
  - check array[index] is empty.
  - store record in array[index].
  - mark location as “taken”.
- To retrieve a record from hash table:
  - calculate index.
  - check array[index] to see if not empty and key matches.
  - return record from array[index] or a “not found” message.
- This approach is very fast but there is one potential problem …

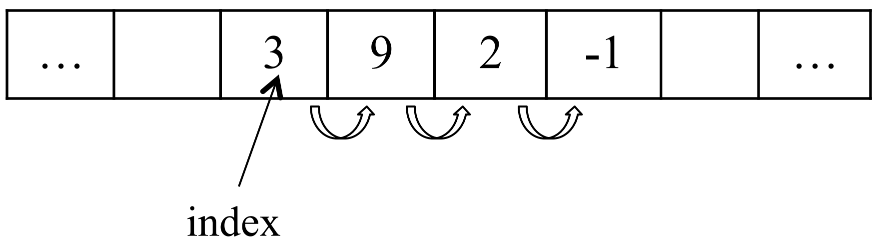
Hashing Collisions

- When we attempt to store a value and the location is already taken this is called a <u>collision</u>.
- Instead of giving up, we must store the data somewhere else in the hash table.
- Many collision resolutions options are possible:
  - Linear probing.
  - Double hashing.
  - Separate chaining.
  - Hash buckets.
- We also need to modify our retrieval algorithm to look in alternative locations if necessary.

Linear Probing

- If the location at index is “taken” simply probe the next locations until an open spot is located.

eg:



- Here we use -1 to mark an open spot in the table.
- New data replaces the -1 and now that location is “taken”.
- We must adapt our lookup to probe until -1 reached to check the locations adjacent to hash table index.
- If data is deleted, we must mark with “deleted” flag and search/insert adjusted accordingly.

Double Hashing

- Similar idea to linear probing except the step size between probes is a function of the hash index.

  eg: step = (index + 79) % 23 + 1

- This will spread out the data records in case of collisions.
- Probes should <u>wrap around</u> at the end of the table (using the modulo % operator).
- Double hashing works best if the step size is relatively prime to table size. Hence use a prime table size.

eg:

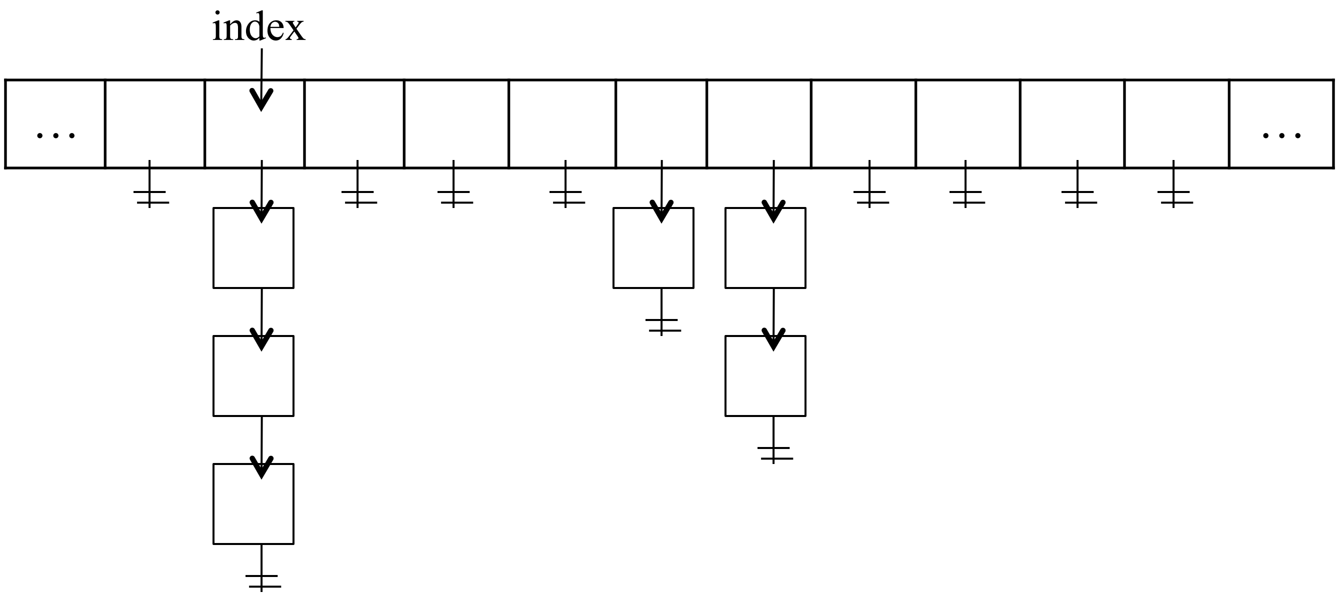
…		3		9		-1	…

index

- Here, index = (index + step) % size

Separate Chaining

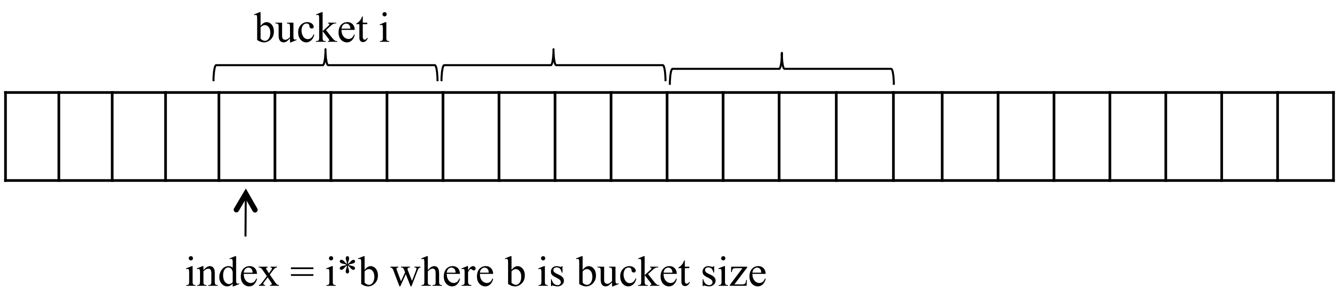
- Instead of probing the hash table we can use a linked list or another dynamic ADT to store collisions.



- Insertion/search/deletion now use our existing linked list code
- Since each list is very small, the operations take very few steps.
- The table size for separate chaining can be smaller because the overflow space is in the linked list.

Hash Buckets

- We can also use fixed size buckets to resolve collisions instead of dynamic data structures.



- The hashing function returns the index of the <u>first</u> location in each bucket.
- We use linear probing to locate empty location in bucket to store data.
- If the bucket becomes full, then we probe next bucket to find empty spot.
- Same spare requirements as linear probing and double hashing.

Hashing Analysis

- Speed of insert/search/delete depends on the number of collisions
- Let α between 0 and 1 be fraction of table that is occupied, hence 1- α is probability location is empty.

Probe	Occupied	Free
1	α	(1-α)
2	α^2	(1-α) α
3	α^3	(1-α) α^2
4	α^4	(1-α) α^3
…	…	..
n	α^n	(1-α) α^(n-1)

- To calculate the <u>average</u> number of probes to locate a free location we sum the product of probe and free columns

Hashing Analysis

$$S = 1(1-\alpha) + 2(1-\alpha)\alpha + 3(1-\alpha)\alpha^2 + \ldots n(1-\alpha)\alpha^{n-1}$$

$$\alpha S = \quad 1(1-\alpha)\alpha + 2(1-\alpha)\alpha^2 + \ldots (n-1)(1-\alpha)\alpha^{n-1}$$

Subtracting and simplifying:

$$(1-\alpha)S = (1-\alpha) + (1-\alpha)\alpha + (1-\alpha)\alpha^2 + \ldots (1-\alpha)\alpha^{n-1}$$

$$S = 1 + \alpha + \alpha^2 + \ldots \alpha^{n-1} = \frac{1}{1-\alpha}$$

<u>examples:</u>

$\alpha = 1/4$, $S = 4/3$

$\alpha = 1/2$, $S = 2$

$\alpha = 3/4$, $S = 4$

$\alpha = 1/10$, $S = 10/9$

- Rule of thumb: make hash tables 2-4 times larger than amount of data you expect to store to keep number of probes small.

Hashing Discussion

- If we wish to store N values in a hash table and we allocate a table of size K*N then the highest value possible for $\alpha=1/K$.
- The average number of probes to find a free location will be:

$$S = \frac{1}{1-\alpha} = \frac{1}{1-1/K} = \frac{K}{K-1}$$

- S does not depend on N, so hashing runs in <u>constant</u> time O(1)
- This is clearly much better than our other ADTs where access time is O(logN) at best.
- Hash tables are widely used in applications when the data can fit in memory.
- Binary search trees are a better choice when data sets are too large for memory and must be stored on disk.